Testing Approaches

Overview

- What is a “Good” test?
- How to design tests?
  - White-box testing
  - Black-box testing

What Is a "Good" Test?

- A good test
  - has a high probability of finding an error
    - Developers must understand the software
  - is not redundant
    - Every test should have a different purpose
  - should be "best of breed"
    - Prioritize tests that have the highest likelihood of uncovering errors
  - should be neither too simple nor too complex
    - Don't try to combine different tests together

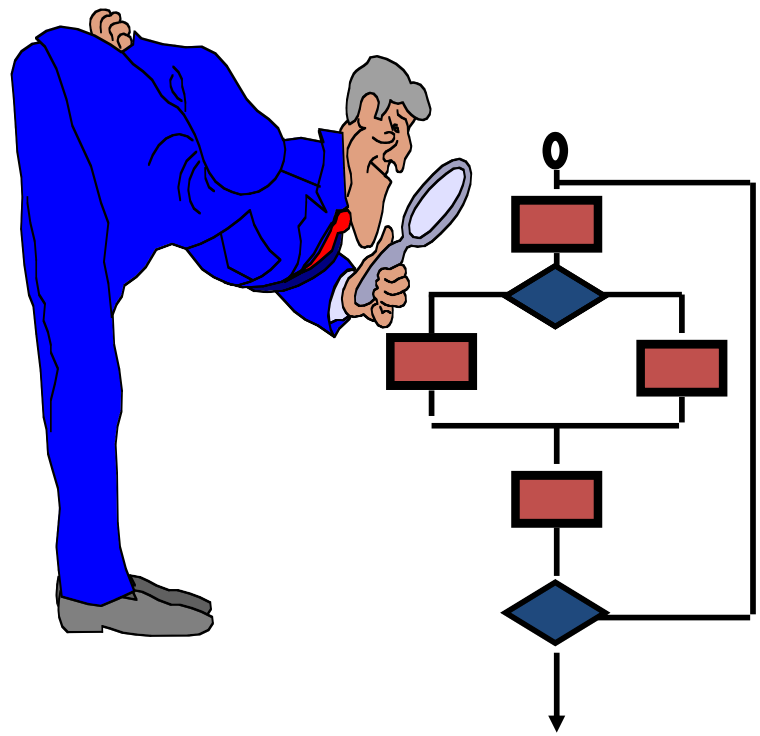
Internal and External Views

- Any engineered product can be tested in two ways:
  - Knowing the internal working of a product, test whether "all gears mesh" and every component has been adequately exercised
  - Knowing the specification, test whether the product conforms to specification

Software Testing Methods

- White-box methods
  - Internal-view approach
- Black-box methods
  - External-view approach

White-Box Testing



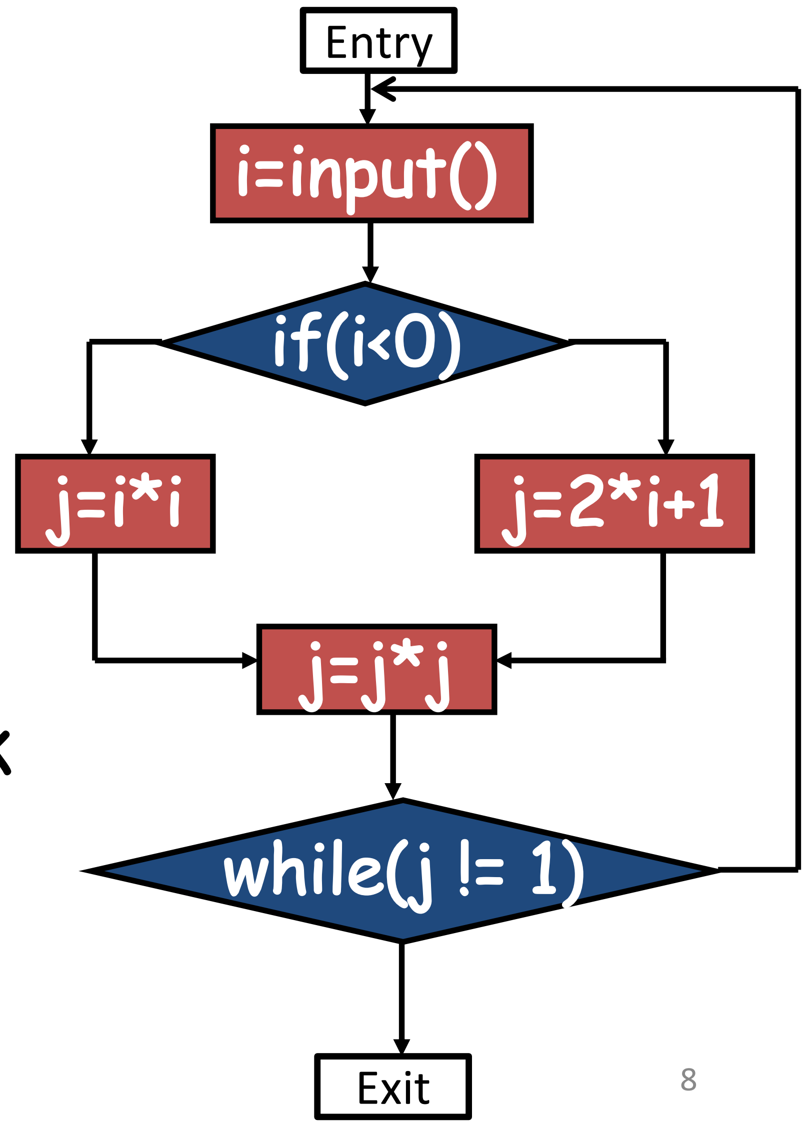
... our goal is to ensure that all statements and conditions have been executed at least once ...

Why Cover?

- Logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- We often **believe** that a path is not likely to be executed; in fact, reality is often counterintuitive

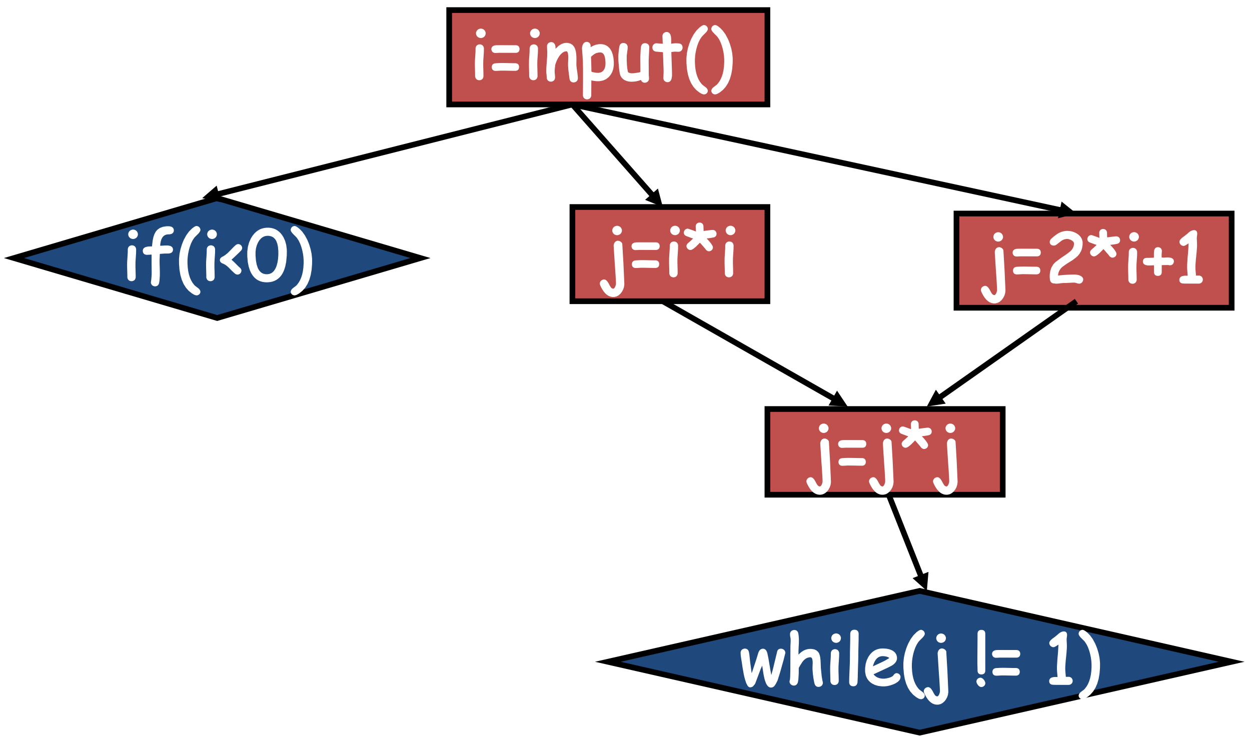
Control Flow Graph

- A representation, using graph notation, of all paths that might be traversed through a program during its execution
  - Node: statement or block
  - Edge: control flow



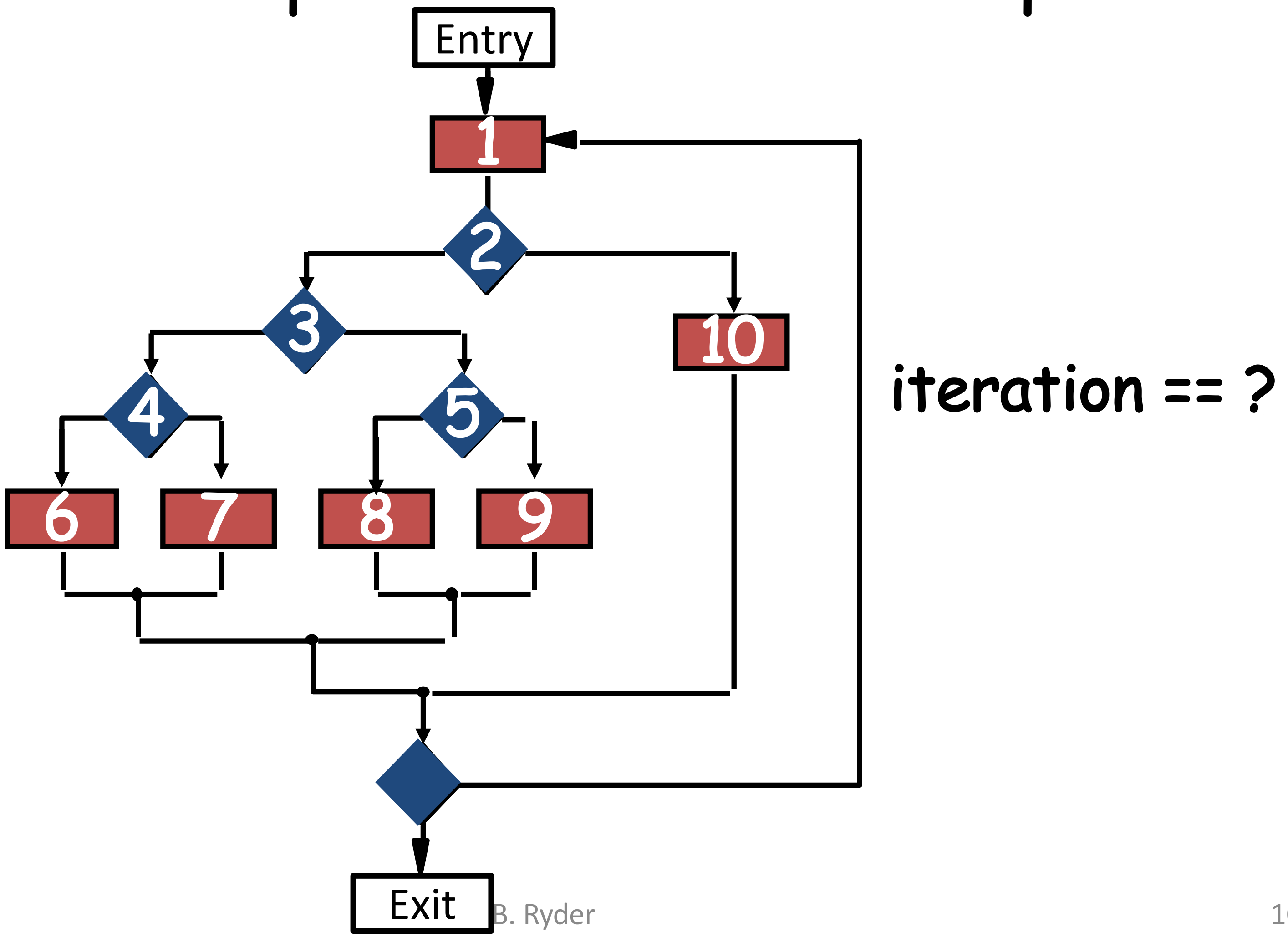
Data Flow Graph

- A representation of the "flow" of data through a system



Naïve Approach: Exhaustive Testing

- Enumerate all possible execution paths



How Many Paths When iteration == 1?

- 5 paths
  - 1,2,3,4,6
  - 1,2,3,4,7
  - 1,2,3,5,8
  - 1,2,3,5,9
  - 1,2,10

How Many Paths When iteration == 20?

- $5^{20} \approx 10^{14}$
- **If we execute one test per millisecond, it would take 3,170 years to test this program!!**

Efficient Approach: Selective Testing

- Control flow-based testing
  - Basis path testing
  - Condition testing
  - Loop testing
- Data flow-based testing

Selective Regression Testing

- Only need to rerun tests which might be affected by program changes
- Idea: do parallel traversal of CFG(P) and CFG(P’): when targets of like-labeled edges differed, then use coverage matrix to find tests that will exercise that edge

Basis Path Testing

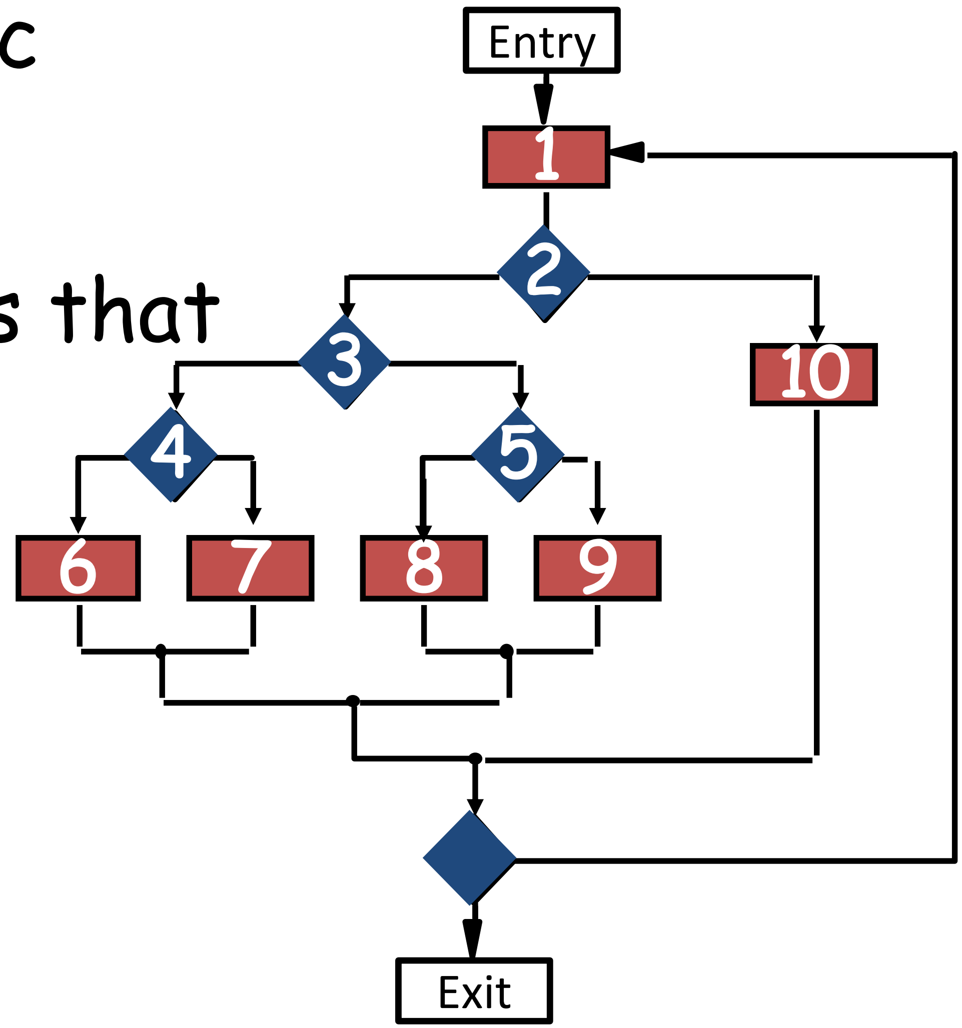
- Independent Path
  - Any path through the program that produces at least one new set of processing statements or a new condition
- To guarantee every statement is executed at least once
  - Statement coverage

Basis Path Testing

- Cyclomatic complexity V(G)
  - number of simple decisions + 1
  - number of enclosed areas + 1
- A number of industry studies have indicated that the higher V(G), the higher the probability of errors.

Basis Path Testing

- What is the cyclomatic complexity?
  - V(G) = 6
- Design V(G) test cases that cover all statements
  - 1,2,3,4,6
  - 1,2,3,4,7
  - 1,2,3,5,8
  - 1,2,3,5,9
  - 1,2,10
  - 1,2,10,1,2,10

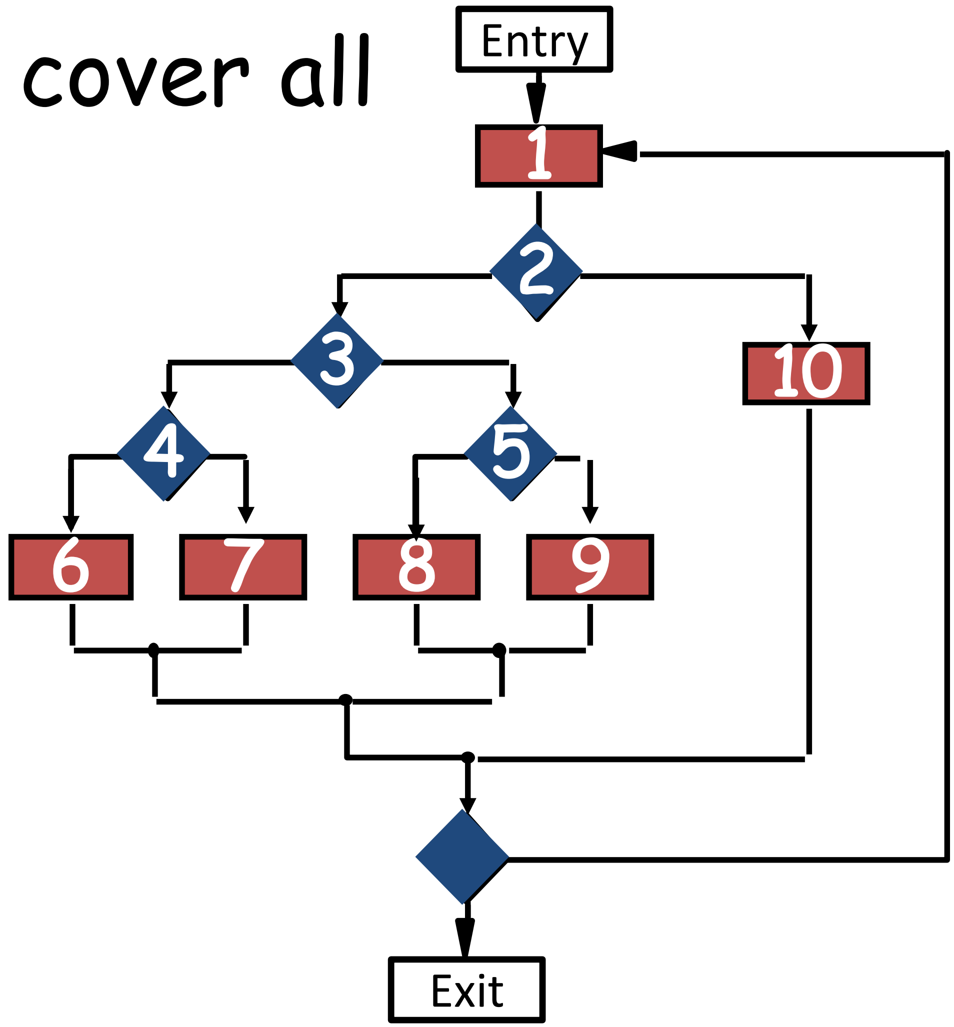


Condition Testing

- To guarantee every branch of the predicate nodes is covered
  - **Branch coverage**
    - True and false branches of each IF
    - The two branches of a loop condition
    - All alternatives in a SWITCH

Condition Testing

- Design test cases to cover all branches
  - 1,2,3,4,6
  - 1,2,3,4,7
  - 1,2,3,5,8
  - 1,2,3,5,9
  - 1,2,10
  - 1,2,10,1,2,10



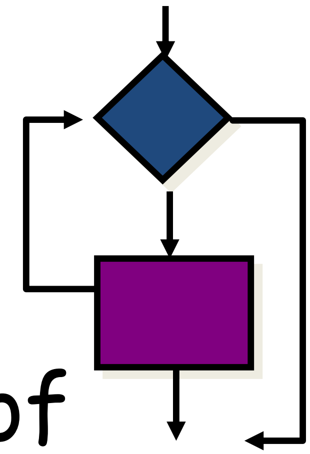
Statement Coverage vs. Branch Coverage

- Branch coverage => Statement coverage, but **not** vise versa
  - E.g., if (c) then s;
    - By executing only with c=true, we will achieve statement coverage, but not branch coverage

Loop Testing

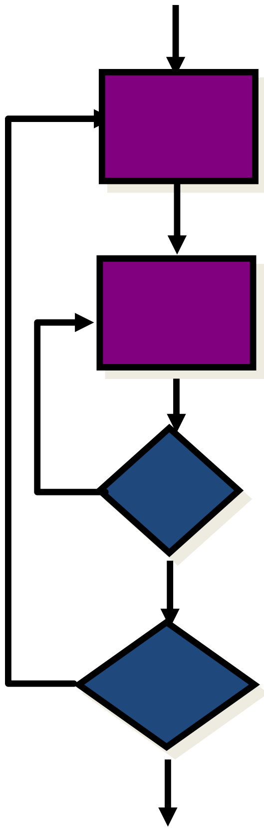
- Test cases only focus on the validity of various loop constructs
  - Simple loops
  - Nested loops
  - Concatenated loops
  - Unstructured loops

Test Cases for Simple Loops



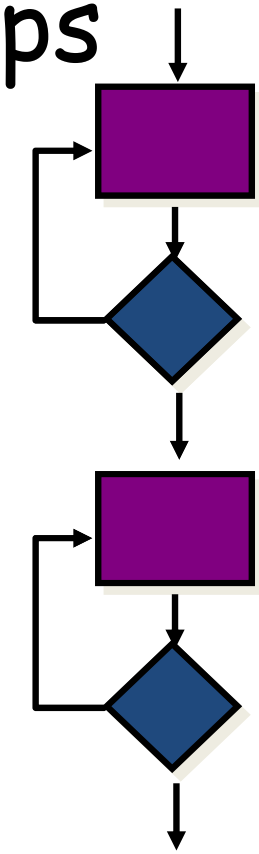
- Suppose n is the maximum number of allowable passes through the loop
  - Skip the loop entirely
  - Only one pass through the loop
  - m passes through the loop where m < n
  - n-1, n, n+1 passes through the loop

Test Cases for Nested Loops



- Suppose the iteration parameter **i** for outer loop is in **[n1, n2]** range, while the parameter **j** for inner loop is in **[m1, m2]**
  - Set i=n1, test inner loop
  - Set j=typical value∈[m1, m2], test outer loop

Test Cases for Concatenated Loops



if (the loops are independent of each other)

then

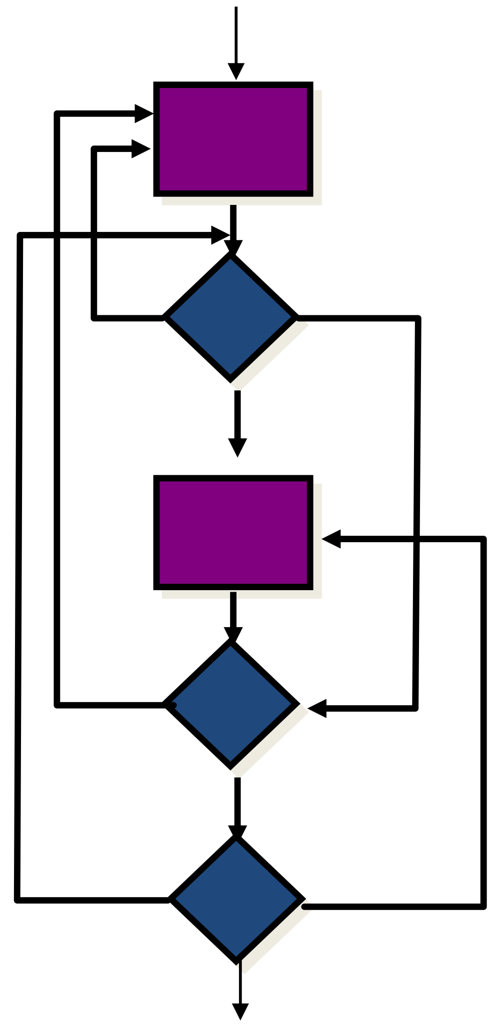
    treat each as a simple loop

else

    treat them as nested loop

Unstructured Loops?

- Whenever possible, redesign!



Homework 3: Testing

```
int gcdByBruteForce(int n1, int n2) {
  if (n1 == 0)
    return n2;
  if (n2 == 0)
    return n1;
  int gcd = 1;
  for (int i = 1; ; i++) {
    if (i > n1)
      break;
    if (i > n2)
      break;
    if (n1 % i == 0) {
      if (n2 % i == 0) {
        gcd = i;
      }
    }
  }
  return gcd;
}
```

Requirements of Test Cases

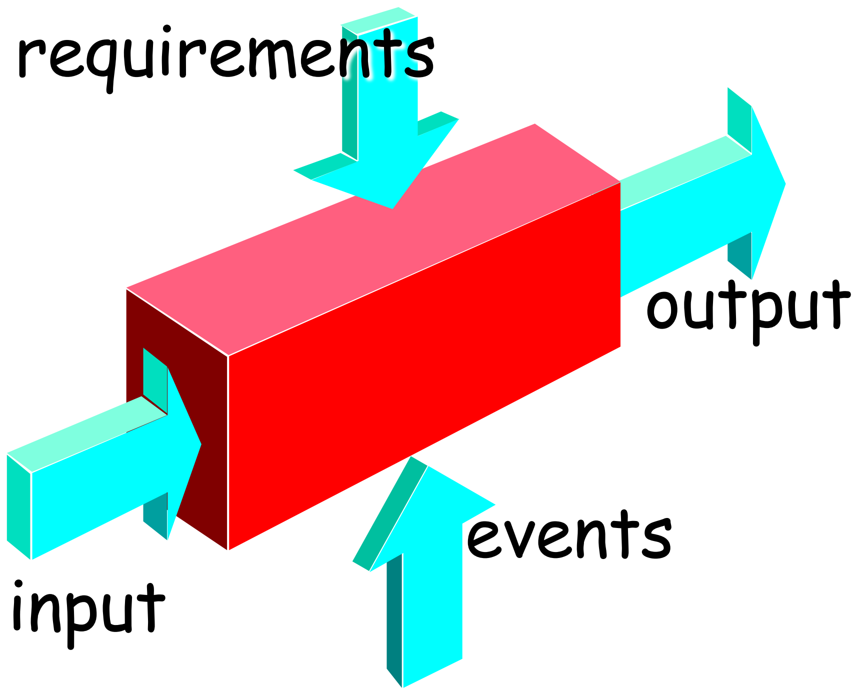
- Draw a CFG, where nodes represent statements, and edges represent the control flow
- Index each CFG node with a number
- Design two sets of test cases to separately achieve
  - Statement Coverage: Ensure that every statement is covered at least once
  - Branch Coverage: Ensure that every branch is covered at least once
- For each designed the test case, describe
  - the test inputs, and
  - the CFG nodes (i.e., the path) covered by the test

- Please organize each set of test cases in a table, as shown below:

n1	n2	path
0	1	1, 2 (here 1 and 2 represent the CFG node indices)
…	…	…

Black-box Testing

- Black-box testing focuses on the software functional requirements
- Testers devise various input conditions to fully exercise all functional requirements



Black-Box vs. White-Box

- Black-box is a complementary approach instead of an alternative to white-box techniques

☐ check "doing the right thing"	☐ check "doing things rightly"
☐ applied during later stages of testing	☐ performed early in the testing process
☐ input-oriented	☐ structure-oriented

Black-Box Methods

- Equivalence partition
- Boundary value analysis

Equivalence Partition

- Divide the input domain of a program into equivalence classes
  - For different values from the same class, the software should behave equivalently
- Test with values from different classes to find errors

How to Define Equivalence Classes?

- An input condition specifies a range
  - Define one valid and two invalid equivalence classes
  - E.g., for input range [2, 5], the equivalent classes are (-∞,2), [2, 5], (5,+∞)
- An input condition specifies a specific value
  - Define one valid and two invalid equivalence classes

How to Define Equivalence Classes?

- An input condition specifies a member of a set
  - Define one valid and one invalid equivalence class
- An input condition is Boolean
  - Classes “true” and “false”

Boundary Value Analysis

- It complements equivalence partition technique by
  - focusing on boundary values of each equivalent class,
  - deriving test cases from the output domain as well

How to Pick Values to Test?

- If an input condition specifies a range [a,b]
  - Design test cases with values a and b and just above and just below a and b
- If an input condition specifies a number of values
  - Design test cases with values min and max and surrounding values
- Apply the above guidelines to output conditions

How to Pick Values to Test?

- If internal program data structures have prescribed boundaries, be certain to design test cases to exercise the data structure at its boundary
  - e.g., a table has a defined limit of 100 entries

Example: Search for a Value in an Array

- Input: an array and a value
- Output: return the index of some occurrence of the value, or -1 if the value does not exist
- One partition: size of the array
  – 0, 1, n (n > 1)
- Another partition: location of the value
  – 0, m(m>0 && m<n), n-1 (last), -1

Example: Test Inputs

Array	Value	Output
empty	5	-1
[7]	7	0
[7]	2	-1
[1,6,4,7,2]	1	0
[1,6,4,7,2]	4	2
[1,6,4,7,2]	2	4
[1,6,4,7,2]	3	-1